Theses of the Ph.D. dissertation

Static and Dynamic Program Analysis

Tamás Gergely

Supervisor
Dr. Tibor Gyimóthy

Ph.D. School in Computer Sciences

University of Szeged
Institute of Informatics

2010

Introduction

The subject of this dissertation is program analysis – detecting relationships between program elements. It can be used for many purposes during the software's life-cycle, for example for test case selection, debugging, error location, program understanding, reverse engineering, or change propagation.

Program analysis is a large and diversified research area, but many of its fields can be distinguished from each other by certain aspects. These aspects include granularity, and whether the method is static or dynamic. The granularity defines whether the relations are defined between high level program elements (e. g. procedures, methods, classes) or between low level elements (source code or assembly instructions, typically). Static analysis can only rely on statically available information (thus without executing the program), while dynamic analysis can use dynamic information acquired during the execution of the program too.

Impact analysis is a high level analysis employing mostly static techniques, while *program slicing* is a low level analysis with both static and dynamic applications. In this dissertation we have discussed these two fields of program analysis. Namely, we present our results of both static and dynamic impact analysis and dynamic program slicing researches.

Our results are summarized in five theses.

- I/1. Definition of *SEA/SEB* relations.
- I/2. Definition, determination and evaluation of *DFC* metric.
- II/1. Determination of *d:U* based slicing algorithms.
- II/2. Implementation of *d:U* based slicing algorithms.
- II/3. Evaluation of *d:U* based slicing algorithms.

I. High level analysis

In several software engineering activities related to software evolution, only certain parts of a system are investigated at a time, and this part of interest may be extended or shifted as the activity progresses. Namely, in a software life-cycle based on incremental change [20], the impact of a change made to the system needs to be determined; this will then be used for change propagation, regression testing and other activities. The key of these activities is to determine the neighborhood of the items.

The notion of such 'neighborhood' may be quite different depending on their actual application scenario. For example, with change propagation a very simple technique is to investigate only the directly dependent classes of the class of interest (according to the class diagram-like relations) in one iteration of the propagation. Similarly, with regression testing a simple, yet very effective technique is based on testing firewalls [26, 27], which means rerunning only test cases that exercise direct (or close) dependents of a changed part.

The aim of impact analysis [11] is to support the different activities of software development and maintenance by determining the impacted program elements. It is usually done by determining different relations between program elements. Different approaches exist to compute relations between higher level software structures supporting impact analysis [4]. Most of the common methods are static, e. g. the work of Rajlich *et al.* or Ren *et al.* [20, 21]. The simplest static methods use the call graph [11] or some other lightweight program dependency relations, which are imprecise or unsafe techniques (e. g. [28]). It is possible to find methods and results that both precise and safe (for example static program slicing), but the computation cost of these methods are found to be too high [14, 24] for impact analysis.

Our methods for computing impact sets on function level are motivated by of the dynamic *Execute After* relation of Apiwattanapong *et al.* [3]. Apiwattanapong *et al.* use a very simple approach that essentially states the following: based on a set of executions, a specific function f will potentially have an impact on all those methods that are executed sometime *after it* in any of the executions, meaning that any function g executed after f will become part of f's impact set. This approach is safe – meaning that no dependence is missed –, but imprecise too. In fact, based purely on the sequence of function calls and returns, it seems to be impossible to provide a more precise, yet still safe method.

I/1. Definition of *SEA/SEB* relations

Some of the dependencies between program components are explicit, for example generalization, composition, association between classes in object oriented systems. Typically these dependencies are expressed in the code as explicit references. However besides explicit dependencies, there are also other dependencies; we call these *hidden dependencies*[1]. Yu and Rajlich [28] explored hidden dependencies that are based on the existence of data flows between otherwise explicitly unrelated components.

We proposed an alternative way to determine the explicit and hidden dependencies by employing *Static Execute After (SEA) relation* and *Static Execute After (SEA) relation* among program components. The SEA is a static counterpart of the approach of Apiwattanapong *et al.* who introduced the notion of Execute After relations [3]. We say that $(f, g) \in SEA$ if and only if any part of g may be executed after any part of f in any of the possible executions of the program. An intrinsic property of the SEA relation is that it is safe but imprecise.

Formally, the SEA/SEB relations can be divided into three (non-distinct) sub-relations:

$$SEA = SEA_{call} \cup SEA_{seq} \cup SEA_{ret} \ ,$$

where

$$\begin{aligned}
(f,g) \in SEA_{call} &\overset{\text{def}}{\iff} f \text{ calls } g,\\
(f,g) \in SEA_{seq} &\overset{\text{def}}{\iff} \exists\, h\text{: } h \text{ calls } f \text{ first, then}\\
&\qquad\ \text{after } f \text{ returned into } h,\ h \text{ calls } g,\\
(f,g) \in SEA_{ret} &\overset{\text{def}}{\iff} f \text{ returns into } g,
\end{aligned}$$

where both 'call' and 'return into' are treated transitively. We also defined the *Static Execute Before* (SEB) relation in a similar way:

$$SEB = SEB_{call} \cup SEB_{seq} \cup SEB_{ret} \ .$$

For computing the SEA relation a suitable program representation is needed. The traditional Call Graph [22] is unsuitable for our needs since it says nothing about the order of the procedure calls within a procedure. On the other hand, an Interprocedural Control Flow Graph (ICFG) [19] contains too much information and is expensive to work with. Thus, we defined a new representation.

First we defined the (intraprocedural) *Component Control Flow Graph* (CCFG), where only nodes and edges important for procedure calls were considered. Each CCFG represents one procedure and contains one *entry node* and several *component nodes* with *control flow edges* connecting them. Furthermore, strongly connected sub-graphs are collapsed into single nodes; this means that if two call sites are reachable from each other by control flow

[1]Note, that these are usually hidden to impact analysis only, a detailed slicing would find most of them

```
e() {
  if(...) {
    f();
  }
  g();
}


f() {
  while(...) {
    h();
    g();
  }
}
```

```
g() {
  if(...) {
    f();
  } else {
    while(...) {
      h();
      if(...) {
        f();
      }
    }
    h();
  }
}

h() {
}
```

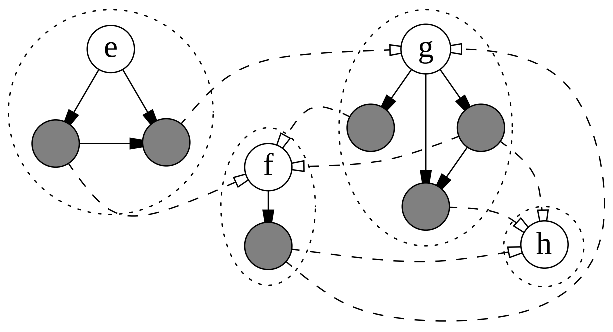

Figure 1: ICCFG example

edges then they are represented by the same component node. *Interprocedural Component Control Flow Graph (ICCFG)* represents the whole system and for each procedure, there is a corresponding CCFG interconnected by *call edges* with other CCFGs. In the ICCFG there is a call edge from a component node c to a procedure entry of m if and only if at least one call site represented by c calls m. An example of ICCFG can be seen in Figure 1.

Own contributions

The definition of *SEA/SEB* relations and the determination of *ICCFG* is a joint contribution. The results were published in [9].

I/2. Definition, determination and evaluation of *DFC* metric

Many of the existing techniques for impact set computation in change propagation and regression testing are approximate for the sake of efficiency. A way to improve precision is to apply dynamic analyses instead of static ones. The dynamic EA relation is also simple and efficient, but yet overly conservative and hence imprecise. The basic idea for refining it is based on the intuition that the 'closer' the execution of a function f is to the execution of function g in some of the runs of the program, the more likely they are dependent on each other.

Before presenting the formal definition, we introduce the concept of *dynamic call tree.* It is a rooted tree with ordered edges, where the node p labeled with f function represents a called instance of the f function, and a $p \to q$ edge represents a function call from the p instance of f to the q instance of g function, where q has a label g. We will also use the term $f \to g$ *call chain*, which is a path from vertex q to vertex p, these being instances of functions f and g, respectively, for which the following holds: the path from the root to q is the prefix of the path from the root to p.

Now, we extend the definition of *Execute After* relation with the measure of indirection level d. Formally:

$$\begin{aligned}
(f,g) \in EA^{(d)}_{call} &\overset{\text{def}}{\iff} \exists\, f \rightarrow g \text{ call chain of length } d,\\
(f,g) \in EA^{(d)}_{ret} &\overset{\text{def}}{\iff} \exists\, g \rightarrow f \text{ call chain of length } d,\\
(f,g) \in EA^{(d)}_{seq} &\overset{\text{def}}{\iff} \exists\, h \text{ function, where:}\\
&\qquad \exists\, h \rightarrow f \text{ call chain of length } d_r \text{ and } h \rightarrow g \text{ call chain of length } d_c\\
&\qquad \text{with only one common point (labeled with } h\text{) in the tree, where}\\
&\qquad f \text{ is called before } g\text{, and } d = d_r + d_c - 1.
\end{aligned}$$

We combine these in the $EA^{(d)}$ relation, which permits the maximal indirection level of d, formally defined as follows:

$$(f,g) \in EA^{(d)} \overset{\text{def}}{\iff} \exists\, d' \leq d \,:\, (f,g) \in EA^{(d')}_{call} \cup EA^{(d')}_{ret} \cup EA^{(d')}_{seq}.$$

Following our view on the symmetry, the *Execute Before* relation ($EB^{(d)}$) can be computed for any d values by replacing the role of the two functions:

$$(f,g) \in EB^{(d)} \overset{\text{def}}{\iff} (g,f) \in EA^{(d)},$$

and by combining these two relations we define the *Execute Round* ($ER^{(d)}$) relation as well, as follows:

$$\forall d : ER^{(d)} = EB^{(d)} \cup EA^{(d)}.$$

Observe, that as special cases of our definitions, $EA^{(\infty)}$ corresponds to Apiwattanapong *et al.*'s definition of the Execute After relation, while $ER^{(\infty)}$ gives the complete graph with the covered functions.

Naturally, if a cut-off level d is sufficient for a pair of functions to be connected by *Execute Round*, all higher levels will be appropriate too. So, the *Dynamic Function Coupling* (DFC) metric defines the lowest d value for each f, g function pairs, for which the two functions are in $ER^{(d)}$ relation:

$$DFC(f,g) = \begin{cases} \min\{d \mid (f,g) \in ER^{(d)}\} & \text{if such } d \text{ exists,}\\ \infty & \text{otherwise.} \end{cases}$$

Observe that $DFC(f,g) = DFC(g,f)$ and $DFC(f,f) = 0$ will be true for any two functions f and g.[2]

Based on the above, for a fixed indirection cut-off value d, the dynamic impact set of a set of changed functions C is the following:

$$ImpactSet^{(d)}(C) = \{g \mid \exists f \in C \,:\, (f,g) \in ER^{(d)}\}.$$

Algorithms

We presented three algorithms; all of them are working on an execution history containing *function entry* and *function return* events.

The first one is a global recursive algorithm, that computes DFC values of all function pairs in $O(t \cdot n^2)$ time in the worst case, and its memory requirement is $O(n \cdot m)$, where n is the number of functions, m is the depth of the call tree, and t is the length of the execution history.

[2]Here we do not follow the traditional convention that a larger value means stronger coupling.

The second one is an on-demand algorithm for impact set computation for a given indirection level d. The worst case time requirement of this algorithm is $O(t \cdot n)$, the memory requirement is $O(n \cdot m)$.

The third algorithm is an on-demand algorithm for impact set computation for a fixed indirection level $d = 1$. It has $O(t \cdot n)$ time and $O(n \cdot m)$ space cost in the worst case. It seems to be the same as the previous algorithm. But while the average cost of the previous one is not much better than the worst case, the time and size requirements of this one reduces to almost $O(t)$ and $O(n + m)$.

Measurements

We made some experiments. We measured the precision and recall of the relations and sub-relations on three open-source Java programs. These values were computed against program slicing results as exact relations. To summarize our findings, we present the answers to the research questions set up before the measurements:

1. *Is it true that a small DFC value between two functions indicates a more probable actual coupling between them?* Yes. Most notably DFC levels 1 or 2 indicate significantly more actual couplings than higher levels.

2. *To what extent does call sub-relation alone and together with the sequence sub-relation reflect actual coupling?* If we observe higher indirection levels, the call sub-relation alone does not represent many of the actual couplings (only about 20% is recalled). This means that a significant part of actual couplings comes from sequence-indirections, so the simple only-call algorithm is not sufficient.

3. *What is the threshold value of parameter d that produces good recall, and what is the precision of the method with that parameter?* The cut-off value of parameter d around 5–15 produces recall near 100%. However precision drops much faster, practically, within 1 or 2 steps it reaches the precision of the original EA method.

4. *What d values should be used when precision is important, and what is the recall in that case?* The best precision values can be obtained at levels one or two. However, the recall is very low in this case.

5. *How much gain can we achieve using this method compared to the original EA relation in terms of the size of the impact sets?* The closest level 1 produces impact sets that are on average 13–15% of the set sizes of the safe method, while level 2 brings in about 25–35%.

Own contributions

The definition, idea of computation and the measurements and evaluation of *DCF* metric are joint contributions. The elaboration of the algorithms computing the *DFC* and the impact sets are my own contribution. The results were published in [6].

II. Low level analysis

Program slicing is both similar to and different than impact analysis. It is similar, because both have the same goal: detect relations between program elements. Yet, it is different, because it provides low level relations and aims precision, but it requires more computations.

Over time, a number of program slicing methods [24, 25] have been elaborated. A significant part of the practical methods compute the slices based on various *dependences* (control- and data-) among the program elements (variables, instructions, addresses, predicates, etc.). The literature is elaborate about the details of static slicing methods. For example, the work by Horwitz *et al.* [14] served as the starting point for a number of subsequent implementations and enhancements, whose basis is the *System Dependence Graph* (SDG).

The basic dynamic slicing methods use different concepts, proposed by researchers like Korel and Laski [17, 18], Agrawal *et al.* [1, 2] and Kamkar *et al.* [16]. The traditional dynamic dependence-based method by Agrawal and Horgan [2] uses a graph representation called the *Dynamic Dependence Graph* (DDG) that includes a distinct vertex for each occurrence of a statement (an action), and the edges correspond to the dynamically occurring dependences. Based on this graph, the computation of a dynamic slice is finding all reachable vertices starting from the slicing criterion.

However, relatively few publications appeared that deal with the practical sides of dynamic slicing and provide detailed algorithms. A reason can be that dynamic analysis of programs is an inherently hard problem because of several reasons, the most significant one being that a very large number of events may be generated by a program run. Most of the basic dynamic slicing algorithms have difficulties with handling large inputs. For example, the size of the DDG graph is actually determined by the number of steps of the execution history which is unbounded.

Tibor Gyimóthy, Gábor Forgács and Árpád Beszédes presented an algorithm for backward slice computation, which computes slices for all dynamic slicing criterion by traversing the execution history only once [13]. Based on this, Árpád Beszédes worked out an algorithm for computing a single slice [5]. These algorithms use so-called $d : U$ (*definition-use*) pairs to represent instructions. Although the original algorithms support only a very simple language, their relatively low space requirements makes them suitable for slicing large programs.

II/1. Determination of *d:U* based slicing algorithms

By examining the two algorithms elaborated by Árpád Beszédes *et al.*, it became obvious that many graph-less slicing algorithms can be created using the same representation. Thus, we determined some aspects of the algorithms, and determined their possible values, and then we examined their combinations. We found three aspects:

Slicing direction. The two slicing directions are *forward* and *backward* slicing. In the case of *forward slicing* we are interested in those program points, that uses (even transitively) the values computed in the program point determined by the slicing criterion. A *backward slice* consists of all statements that might affect the values computed at a specific program point.

Global or demand-driven. In the traditional approach we have one criterion at a time, and we compute slice for this criteria. It is called *demand-driven* slicing. However, it is possible to compute more (or all possible) slices by traversing the execution history only once. In this case we are talking about *global* slicing.

Processing direction. The execution history can also be processed in two ways. The *forward processing* is the 'natural' direction, as the execution history is generated this way. Sometimes this is the only feasible direction. However, there are situations when the *backward processing* can be applied and more efficient than the other direction.

Global/Demand-driven	Slicing direction	Processing direction	Usability
Demand-driven	backward	backward	practical
Demand-driven	backward	forward	unfeasible
Demand-driven	forward	backward	unfeasible
Demand-driven	forward	forward	practical
Global	backward	backward	parallel
Global	backward	forward	practical
Global	forward	backward	practical
Global	forward	forward	parallel

Table 1: Overview of dynamic slicing algorithms

This totals eight possibilities, of which some give useful algorithms, while there are irrelevant combinations as well. These are summarized on Table 1.

Computing a dynamic slice in a demand-driven fashion means that given an execution of the program and a dynamic slicing criterion, a single dynamic slice is produced. We traverse the execution trace starting with the action of the dynamic slicing criterion, and follow the dynamic dependences with the help of the $d : U$ representation going backward towards the first executed instruction or forward towards the end of the trace, depending on the slice direction. This allows us to construct the two demand-driven dynamic slicing algorithms.

Computing slices in a demand-driven fashion with opposite slicing and processing direction is unfeasible. It practically results in a global algorithm, because all slices must be kept until the criterion is reached in the execution history.

In a number of applications more than one slice may be needed at a time for a given execution of the program. This leads to an idea to compute more dynamic slices during only one traversal through the execution history. It is possible to compute many dynamic slices by executing the demand driven methods in parallel: traversing the execution history in a forward way for forward slices and in a backward way for backward slices. However, this approach is not very practical since the data structures (and the slices) for all dynamic criteria need to be maintained throughout the whole execution history.

Fortunately, it is possible to construct such global algorithms that are more practical in which not the whole dynamic slices need to be maintained during the execution of the algorithms but only the actual dependence sets belonging to the variables of the program. These dependence sets contain statement numbers providing the actual dependences of the given variables at the given point of execution. We derive these dependence sets based on the $d : U$ information and maintain them for each execution step. Thus we are able to compute the dynamic slices for all possible dynamic criteria based on the actual values of these sets only. An interesting duality in this approach is that the mentioned dependence sets can be acquired the trace is processed in an opposite direction as the slicing direction.

DDG equivalence

To show that our $d : U$-based algorithms compute the same slices as the DDG-based method, we must show the equivalence of the two representations first. Let the instructions of the program be identified with the numbers $i \in \{1, \dots, I\}$. Given the PDG (*Procedure Dependency Graph*, a component in the SDG) and $d : U$ representation of the same program. By definition:

$$\begin{array}{rcl}
\text{in } \mathbf{PDG} & & \text{in } d:U \\
\exists P_i & \Longleftrightarrow & \exists d_i : U_i \\
\exists P_i \rightarrow P_k \text{ control flow edge} & \Longleftrightarrow & d_i \text{ predicate variable} \in U_k \\
\exists P_i \rightarrow P_k \text{ data flow edge} & \Longrightarrow & d_i \in U_k
\end{array}$$

Based on the above, it can be shown that a $P_{i^j} \rightarrow P_{k^l}$ edge exist in the DDG if and only if $d_i \in U_k$ and the d_i variable is last defined in i^j before the l step $(LD(d_i, l) = i^j)$.

In the case of demand-driven algorithms, the equivalence was shown through transforming the algorithm into a graph-coloring algorithm using equivalent transformations. It colors the points of the graph reachable from a start node. Instructions belonging to the colored points form the DDG-based slice.

To show that the global algorithms compute the DDG-based slices we use induction. In the beginning the sets used by the algorithms are empty, which is trivially correct before processing any trace elements. Then suppose that at the beginning of the iteration that processes i^j the sets contain instructions or actions that are reachable from the corresponding DDG-node. Finally, regarding the algorithms we show that if the assumptions are correct before processing i^j then they remain correct after processing it too.

Own contributions

Classification of the algorithms, the elaboration of the four new algorithms (demand driven forward, global backward processed forward slices, and the two parallel) are joint contributions. Showing that the slices are equivalent to the DDG-based slices is my own contribution. Results were published in [7] and in the technical report [8].

II/2. Implementation of *d:U* based slicing algorithms

We implemented the dynamic slicing algorithms for C and Java languages. To slice real C programs several problems, such as *pointers*, *function calls* and *jump statements* must be solved. As a first step, we modified the $d : U$ representation according to the need of representing C instructions. For C programs, the $d : U$ representation will contain a *sequence* of $d : U$ *items* for each instruction as:

$$i.\ \langle (d_1 : U_1), (d_2 : U_2), \ldots \rangle\ .$$

The sequence order is important and determined by the 'execution-order' of the corresponding sub-expressions.

The execution history was also modified. We added some technical information to it, like memory addresses, block entry/exit events, function call/return events, etc. This extended *EH* is called *TRACE*. The *TRACE* is produced by first instrument the program (adding instructions to it), then executing the instrumented version.

The handling of pointers was solved by converting everything to memory locations (when it is possible). Thus, during the algorithm's execution, we need another $d : U$ structure, called *dynamic* $d : U$. This *dynamic* $d : U$ contains the memory locations, and the algorithm works on it.

Handling pointers

The address of a variable does not change in its scope, so after it is determined it can be used any number of times. But the value of a pointer can change at any time and must be determined every time the pointer occurs. Thus, the instrumented program writes these addresses into the *TRACE* using the functions `remember()` for variables and `dump()` for pointers:

```
    | int x, *p;  | int x, *p;
    |             | remember("x", &x, sizeof(int));
    |             | remember("p", &p, sizeof(int*));
 1. | x=1;        | x=1;
 2. | p=&x;       | p=&x;
 3. | *p=2;       | *dump("PTR1", p,, sizeof(int))=2;
 4. | print(x);   | print(x);
```

The static and dynamically resolved $d : U$ of the program and the computed slices for line 4. – provided that means that variables x and p has the addresses `01` and `02` – are the following:

line	def	:	USE	action	def	:	USE	$Slice$
1	x	:	$\emptyset$	1^1	`01`	:	$\emptyset$	$\emptyset$
2	p	:	$\emptyset$	2^2	`02`	:	$\emptyset$	$\emptyset$
3	$PTR1$	:	$\{p\}$	3^3	`01`	:	$\{$`02`$\}$	$\{2\}$
4	OUT	:	$\{x\}$	4^4	OUT	:	$\{$`01`$\}$	$\{2,3\}$

In the C language the arrays and the pointers are practically the same and the conversion from one to the other is quite simple. The i^{th} element of an array `t`, denoted by `t[i]`, can be expressed as a pointer `*(t+i)`. Then, when an element of an array is referenced, it is treated as a pointer in the $d : U$ and then its address is written out.

The offset of the members of a structure could be determined statically but the computation of dynamic addresses would be quite complicated. Instead, the members of a structure will also be treated as pointers. In this way the structure member accesses are reduced to pointers. The structures themselves are not converted; they are handled as regular variables.

The address itself does not correctly describe a variable. For example the address of a struct and its first member are the same, but assigning a new value to a whole structure indicates dependencies through all its members. Thus, sizes are also recorded by `remember()` and `dump()`.

Algorithm

Our method for slicing C programs works as follows. First, the input program is analyzed and instrumented, and the static $d : U$ representation is built. Next, the instrumented program is compiled and executed to produce the *TRACE*. Finally, the dynamic slice algorithm is executed using the previously created $d : U$ representation and *TRACE*.

To handle the *TRACE* and convert variables to memory addresses, the *TRACE* handling loop of the algorithms are modified as follows. Based on the type of the actual *TRACE* element, the following actions must be taken.

- *function begin mark*: The processing of the actual $d : U$ item is suspended and the position is pushed on a stack.
- *function end mark*: The processing continues at the $d : U$ position saved on the top of the stack. The value is removed from it.

- *EH element*: The current action will be the one specified by the element, the processing continues with its first $d : U$ item.
- *other*: The unresolved references are translated to memory addresses based on this element.

The variables of the static $d : U$ are resolved in the dynamic $d : U$ according to their type:

- *Scalar variables.* They have a constant address in the scope where they are declared. The addresses are resolved by simulating the stack of the C program (using addresses and block entry/exit events). The dynamic $d : U$ uses the addresses.
- *Dereference variables.* Denoted by $PTRn$, where n is a global counter for each dereference occurrence. They can be resolved directly from the $TRACE$.
- *Predicate variables.* Denoted by Pn, where n is the serial number of the predicate instruction. The depth of the program call stack is appended to them in the dynamic $d : U$, to avoid collisions due to recursive calls.
- *Output variables.* Denoted by $OUTn$, where n is the instruction number. Output variables are a kind of 'dummy' variables that are generated at those places where a set U is used but no other variable takes any value from it. They remain unchanged in the dynamic $d : U$.
- *Function call argument variables.* Variables denoted by $ARG(f, n)$, where f is a function name and n is the function argument (parameter) number. An argument variable is *defined* at the function call site and *used* at the entry point of the function. They remain unchanged in the dynamic $d : U$.
- *Function call return variables.* Denoted by $RET(f)$, where f is a function name. A return variable is *defined* at the exit point of the function and *used* at the function caller after returning. They remain unchanged in the dynamic $d : U$.

After it, if the actual dynamic $d : U$ item can be processed (e. g. contains no unresolved variables) then it is processed.

Own contributions

The handling of variables in the C slicing algorithm, thus assigning source code references and runtime memory addresses is my own result. The results were published in [10] and [12].

II/3. Evaluation of *d:U* based slicing algorithms

We made two kinds of evaluation. First, we analyzed the complexities of our six slicing algorithms and compared them to the DDG based method. Second, we made different measurements with the C and Java implementations.

Complexities

When time and space requirements are elaborated, we concentrate on the core of the algorithms. For example, the reading and storing of the trace or the building and storing of the static representation of the program are not counted. We also omit modifications required for a certain implementation.

Algorithm	time maximum	time average
Demand-driven backward	$J \cdot V \cdot \log(J)$	$J + DEP \cdot \log(J)$
Demand-driven forward	$J \cdot V$	J
Practical algorithms	$J \cdot I \cdot V \cdot \log(I)$	$J \cdot DS \cdot \log(DS)$
Parallel algorithms	$J^2 \cdot (\log(I) + V \cdot \log(J))$	$J \cdot DEP \cdot \log(DS \cdot DEP)$
DDG: one slice	$J \cdot V$	DEP
DDG: building and one slice	$J \cdot V$	$J + DEP$
DDG: all slices	$J^2 \cdot V$	$J \cdot DEP$

Table 2: Computational complexities of slicing algorithms

Algorithm	space maximum	space average
Demand-driven backward	J	J
Demand-driven forward	V	V^{DEF}
Practical algorithms	$V \cdot I$	$V^{DEF} \cdot DS$
Parallel algorithms	$J \cdot (I + V)$	$J \cdot DS + V^{DEF} \cdot DEP$
DDG	$J \cdot V$	J

Table 3: Space complexities of slicing algorithms

We summarized the computational and space requirements of the six $d : U$-based and the DDG-based algorithms on tables 2 and 3. The notations we used: J is the length of the execution history; I is the number of instructions; V is the number of program variables; V^{DEF} is the number of defined variables during program execution; DS is the average slice size; DEP is the average number of points reachable from a certain node in the DDG. (DS is related to I, while DEP is related to J.) Values presented for demand-driven algorithms denote the computation of one slice only, while values presented for practical and parallel algorithms denote the computation of all slices.

In a general case our demand-driven algorithms can be more effective than the DDG-based method, because they determine the slices while traversing (a part of) the EH only once, and does not require a separate full traversal, which is needed in the DDG-based method. Moreover, the number of dynamic dependences kept in the memory at a time is limited, thus their space requirements are also smaller (and it is true not only for the forward slicing, where it is obvious due to $V^{DEF} \leq J$).

The time requirements of the practical algorithms in a general case are not better or worse than that of the DDG-based method. As DS is related to the size of the program it is bounded, while DEP is related to the length of the EH, thus potentially unbounded, with a suitably long execution trace the $d : U$ based method seems to be more practical. The $O(V^{DEF} \cdot DS)$ space requirements of our algorithms are more practical then the $O(J)$ space requirement of the DDG-based method in a general case (taking into consideration that in real applications the value of V^{DEF} is rather dependent on V than on J).

However, our parallel algorithms are obviously worse than the DDG-based method regarding both time and space requirements.

Measurements with the C implementation

The aim of the measurements made with the C implementation was the verification of the algorithms' practical usability. We made experiments with the demand-driven and practical backward implementations on five small programs: *bcdd*, *unzoo*, *bzip*, *bc*, and *less*. During the measurements, we recorded some properties of the test programs and the algorithms.

Our findings were:

- No relation was found between the slice size and the length of the execution history.
- The correlation between the number of static variables (in the program code) and dynamic variables (allocated during execution) is relatively high 0.73. Based on the results replacing the variables to memory locations causes no problem resulting from the multiplication of the number of variables.
- For practical usability, the relation between the number of set operations and the size of the program or the length of the execution history is important. The maximal size of the sets and the average number of set operations per step are changed more or less together with program size. We also found, that the maximal set size did not grow significantly with the progress of processing the trace.
- In case of the demand-driven algorithm a longer execution history did not imply the growth of the algorithm's iteration steps.
- The size of the set that influences the number of iterations of the demand-driven algorithm was highly correlated with the slice sizes.

As a summary, we can conclude that factors that determine the execution time of the algorithms (number of dynamic variables or set operations) are mainly depend on static components, and the number of iteration steps of the demand-driven algorithm is much smaller than the length of the *EH*.

Measurements with the Java implementation

The measurements made with the Java implementation were focused on different slice sizes, namely the relation between static, dynamic and union slices were measured. Static slices were computed using the *Indus* [15] Java static slicer.

Our measurements were made on five small open source Java programs (*RayTracer*, *JSubtitles*, *NanoXML/DumpXML*, *java2html*, and *dynjava*) with about 100 test cases per program. Statistics on the number of executed instructions can be seen on Table 4.

	Executed instructions	
Program	minimum	maximum
RayTracer	2,598,546	21,525,307,460
JSubtitles	516,213	55,459,126
NanoXML	910,806	94,754,237
java2html	1,541,531	20,370,505
dynjava	4,019,365	6,369,636

Table 4: Executed instructions

Our findings were:

- Union slices are much smaller than static slices.
- The sizes of the forward union slices are lower than the sizes of the backward union slices.

- The number of smaller slices among the forward slices is higher, but the maximal sizes more or less the same as the maximal sizes of the backward slices.
- The correlations between union slice sizes and instruction coverage are between 0.89 and 0.96. It is good, because coverage can be exactly determined, thus the final slice size can be approximated.

So, important results are that the sizes of the union slices are much lower than static slice sizes, and that the growth of the union slices (by adding more and more dynamic slices) are highly correlated with the instruction coverage growth.

Own contributions

The evaluation of the theoretical algorithms is my own result, which was summarized in [7] and elaborated in a technical report [8]. The evaluation of the C implementation is a joint work, published in the papers [10], [12] and in the report [8]. The evaluation of the measurements made with the Java implementation is a joint work too, and it was published in [23].

Acknowledgements

The dissertation summarizes results that were the results of a team work. Although, sometimes the boundaries of the individuals' works are clear, the results are hard to be interpreted separately. I would like to thank Tibor Gyimóthy for his guidance, co-authors of our papers Árpád Beszédes, Csaba Faragó, Gabriella Tóth, Judit Jász, Ferenc Fischer, Zsolt Szabó, Attila Szegedi, Szabolcs Faragó, Václav Rajlich, János Csirik, and my colleagues Ferenc Havasi, Ákos Kiss, László Vidács, István Siket, Rudolf Ferenc and those I did not mention here by name for the joint work.

I would like to thank my family, my wife, my children and my parents for their patience and support.

References

[1] Hiralal Agrawal. *Towards Automatic Debugging of Computer Programs.* PhD thesis, Purdue University, 1992.

[2] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, number 6 in SIGPLAN Notices, pages 246–256, White Plains, New York, June 1990.

[3] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, pages 432–441, May 2005.

[4] Robert S. Arnold. *Software Change Impact Analysis.* IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[5] Árpád Beszédes. *Forráskód Analízis és Szeletelés a Programmegértés támogatásához.* PhD thesis, Szegedi Tudományegyetem, Matematika- és Számítástudományok Doktori Iskola, Szeged, November 2004.

[6] Árpád Beszédes, Tamás Gergely, Szabolcs Faragó, Tibor Gyimóthy, and Ferenc Fischer. The dynamic function coupling metric and its use in software evolution. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR 2007)*, pages 103–112. IEEE Computer Society, March 21–23, 2007.

[7] Árpád Beszédes, Tamás Gergely, and Tibor Gyimóthy. Graph-less dynamic dependence-based dynamic slicing algorithms. In *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006)*, pages 21–30. IEEE Computer Society, September 27–29, 2006.

[8] Árpád Beszédes, Tamás Gergely, and Tibor Gyimóthy. Investigation of graph-less dynamic program slicing algorithms. Technical report, University of Szeged, 2007.

[9] Árpád Beszédes, Tamás Gergely, Judit Jász, Gabriella Tóth, Tibor Gyimóthy, and Václav Rajlich. Computation of static execute after relation with applications to software maintenance. In *Proceedings of the 23rd IEEE International Conference on Software Maintenance (ICSM 2007)*, pages 295–304. IEEE Computer Society, October 2–5, 2007.

[10] Árpád Beszédes, Tamás Gergely, Zsolt Mihály Szabó, János Csirik, and Tibor Gyimóthy. Dynamic slicing method for maintenance of large C programs. In *Proceedings of the 5th European Conference on Software Maintenance and Reengineering (CSMR 2001)*, pages 105–113. IEEE Computer Society, March 14–16, 2001. Best paper of the conference.

[11] Shawn A. Bohner and Robert S. Arnold, editors. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[12] Csaba Faragó and Tamás Gergely. Handling pointers and unstructured statements in the forward computed dynamic slice algorithm. *Acta Cybernetica*, 15:489–508, 2002.

[13] Tibor Gyimóthy, Árpád Beszédes, and István Forgács. An efficient relevant slicing method for debugging. In *Proceedings of the Joint 7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE'99)*, number 1687 in Lecture Notes in Computer Science, pages 303–321. Springer-Verlag, September 1999.

[14] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.

[15] Indus project: Java program slicer and static analyses tools. `http://indus.projects.cis.ksu.edu/`.

[16] Mariam Kamkar, Nahid Shahmehri, and Peter Fritzson. Interprocedural dynamic slicing. In *Proceedings of the 4th International Conference on Programming Language Implementation and Logic Programming (PLILP'92)*, volume 631 of *Lecture Notes in Computer Science*, pages 370–384. Springer-Verlag, 1992.

[17] Bogdan Korel and Janusz W. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, October 1988.

[18] Bogdan Korel and Janusz W. Laski. Dynamic slicing in computer programs. *The Journal of Systems and Software*, 13(3):187–195, 1990.

[19] William Landi and Barbara G. Ryder. Pointer-induced aliasing: a problem taxonomy. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 93–103. ACM Press, January 1991.

[20] Václav Rajlich and Prashant Gosavi. Incremental change in object-oriented programming. *IEEE Software*, 21(4):62–69, 2004.

[21] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara Ryder, and Ophelia Chesley. Chianti: A tool for change impact analysis of Java programs. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'04)*, pages 432–448, October 2004.

[22] B. G. Ryder. Constructing the Call Graph of a Program. *IEEE Transactions on Software Engineering*, SE-5(3):216–226, May 1979.

[23] Attila Szegedi, Tamás Gergely, Árpád Beszédes, Tibor Gyimóthy, and Gabriella Tóth. Verifying the concept of union slices on Java programs. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR 2007)*, pages 233–242. IEEE Computer Society, March 21–23, 2007.

[24] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[25] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984.

[26] Lee White and Khalil Abdullah. A firewall approach for the regression testing of object-oriented software. In *10th International Software Quality Week (QW'97)*, page 27, May 1997.

[27] Lee White, Khaled Jaber, and Brian Robinson. Utilization of extended firewall for object-oriented regression testing. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 695–698, September 2005.

[28] Zhifeng Yu and Václav Rajlich. Hidden dependencies in program comprehension and change propagation. In *Proceedings of the 9th International Workshop on Program Comprehension (IWPC'01)*, pages 293–299, May 2001.